# CS 342 Design Document
# Project 3 A, B, C – Virtual Memory

Shrinivas Acharya
10010164

---

------ DATA STRUCTURES ------
Added the following data structures:

Structure for the frame table. It has the following enteries:

1.) v_addr: This is the virtual address of the page which resides at this particular frame.
2.) tid: Thread id of the thread occupying the frame. Together with the previous entry, these two fields uniquely identify the process and the page occupying the frame.
3.) Occupied: Is the frame occupied by some process. This field would be required in page eveiction algorithm to check whether the main memory is full, or some frames are un-occcupied, i.e. have occupied as false.

```
struct fte
{
        uint32_t v_addr;
        tid_t tid;
        bool occupied;
};
```

Structure for the supplementary page table. It entries are as follows:

1.) v_addr: The virtual addres to which this page table entry corresponds to.
2.) type: Required to identify the location of this page, 0 means in memory, 1 means in file (secondary memory), 2 means all zeros (for heap and data segments initial states) and 4 means in the swap area.
3.) writable: Is the page read/write or read only.
4.) file: In case the page resides in the secondary memory, this identifies the file pointer.
5.) ofs: Identifies the offset of the page in the file.
6.) read_bytes: No of bytes to be read from the file starting from offset, while bringing in this page.
7.) zero_bytes: No of bytes to be filled with zero, while bringing in this page.
8.) swap_slot: Identifies the swap slot where the page is present. Data type may need to be changed according to the implementation of swap area (to be done in project 3-D).
9.) elem: To store this structure as a part of the list of every thread.

```
struct spt
{
        uint32_t v_addr;
        int type;                       /* 0:in memory, 1:in file, 2:all zeroes, 3:in swap area */
        bool writable;
        struct file *file;
        int32_t ofs;
        uint32_t read_bytes;
        uint32_t zero_bytes;
        int swap_slot;          //*May need to change to some other data type later on*/
        struct list_elem elem;
};
```

The following declaration is used to create the frame table in thread.h, and each occupied entry of the frame table members is set to false in main() function when PintOS starts.

struct fte frame_table[1024];

The list implementation of the supplementary page table for each thread is added in the struct thread:

struct list sup_pt; /* Supplementary page table per process */

This is initialized along with other lists in init_thread() function in thread.c

------ ALGORITHMS ------

In the original implementation of PintOS, all the pages of the programs loadable were brought in at the beginning when the process started, in the load() function which called the load_segment() to load the various segments of the process.

In the new implementation, instead of calling load_segment(), I make a call to a function fill_spt() implemented by me, which fills the supplementary page table entries of the process with the page information of all the segments of the process. Rest all is kept the same. Then when the process starts, it page faults because the requires page is not found in the main memory. Here in the page fault handler in exception.c, I make use of the supplementary page table to get the location of the page. For now the pages are only in file or all-zero. In either case, I call the load_segment() function with the required data items stored in the supplementary page table (like file, offset, read_bytes, zero_bytes and writable). The load segment then loads the page in the main memory. Thus the page fault is handled.

Now there is a need to fill the frame table accordingly and to update the supplementary page table entry corresponding to this page. This is done in the install_page() function which is called by the load_segment() function for successful installation of the page in the memory. Here the frame tabel and supplementary page table are updated accordingly.

There is also the requirement of updating the frame table and supplementary page table when the stack is set up. While setting up the stack as well the install_page() function is called, and hence no extra code is required.

------ SYNCHRONIZATION ------

There were no special race conditions to consider in this project and hence no extra synchronization efforts were required.

------ RATIONALE ------

1.) The advantage of the implementation is that updating page table and frame table is done in the install_page() function which is sure to be called while loading any page in the memory and hence there is no need to consider different places from where the page might be loaded.
2.) Another advantage is of using a list implementation of the supplementary page table rather than an array, thereby using only as much memory space as required, since not all the virtual pages are used by the process and hence storing them in an array is clear wastage of space.
3.) The list implementation has a disadvantage that it has time consuming operations like search. A hash-table implementation could have been better in this regard.